java.util.Stream

支持对元素流进行函数式操作的类，例如集合上的 map-reduce 转换。例如：

```
int sum=widgets.stream()
        .filter(b->b.getColor()==RED)
        .mapToInt(b->b.getWeight())
        .sum();
```

这里我们使用widgets ，一个Collection ，作为流的源，然后对流进行filter-map-reduce，得到红色widget的权重之和。 （求和是归约操作的一个例子。） 这个包中引入的关键抽象是stream 。 Stream 、 IntStream 、 LongStream和DoubleStream类是对象和原始int 、 long和double类型的流。流在以下几个方面与集合不同：

- 没有存储。流不是存储元素的数据结构；相反，它通过计算操作的管道从数据结构、数组、生成器函数或 I/O 通道等源传送元素。
- 本质上是功能性的。对流的操作会产生结果，但不会修改其源。例如，过滤从集合中获得的Stream会生成一个没有过滤元素的新Stream ，而不是从源集合中删除元素。
- 懒惰寻求。许多流操作，例如过滤、映射或重复删除，可以延迟实现，从而为优化提供机会。例如，“查找具有三个连续元音的第一个String ”不需要检查所有输入字符串。流操作分为中间（Stream产生）操作和终端（产生价值或副作用）操作。中间操作总是懒惰的。
- 可能无界。虽然集合的大小是有限的，但流不需要。诸如limit(n)或findFirst()之类的短路操作可以允许对无限流的计算在有限时间内完成。
- 消耗品。流的元素在流的生命周期中只被访问一次。像java.util.Iterator一样，必须生成一个新流来重新访问源的相同元素。

可以通过多种方式获得流。一些例子包括：

1. 从java.util.Collection通过stream()和parallelStream()方法；
2. 从数组通过java.util.Arrays.stream(Object[]) ；
3. 来自流类的静态工厂方法，例如Stream.of(Object[]) 、 IntStream.range(int, int) 或Stream.iterate(Object, UnaryOperator) ；
4. 文件的行可以从java.io.BufferedReader.lines()获得；
5. 文件路径流可以从java.nio.file.Files中的方法获得；
6. 可以从java.util.Random.ints()获得随机数流；
7. JDK 中的许多其他流承载方法，包括java.util.BitSet.stream() 、 java.util.regex.Pattern.splitAsStream(CharSequence)
8. java.util.jar.JarFile.stream() 。

流操作和管道

流操作分为中间操作和终端操作，组合起来形成流管道。流管道由源（例如Collection 、数组、生成器函数或 I/O 通道）组成；后跟零个或多个中间操作，例如Stream.filter或Stream.map ；以及诸如Stream.forEach或Stream.reduce之类的终端操作。

中间操作返回一个新流。他们总是很懒惰；执行诸如filter()之类的中间操作实际上并不执行任何过滤，而是创建一个新流，当遍历该流时，它包含与给定谓词匹配的初始流的元素。直到管道的终端操作被执行，管道源的遍历才开始。

终端操作，例如Stream.forEach或IntStream.sum ，可能会遍历流以产生结果或副作用。终端操作执行后，流管道被视为消耗，不能再使用；如果需要再次遍历同一个数据源，则必须返回数据源获取新的流。在几乎所有情况下，终端操作都是急切的，在返回之前完成对数据源的遍历和管道的处理。只有终端操作iterator() 和spliterator()不是；这些是作为“逃生舱”提供的，以在现有操作不足以完成任务的情况下启用任意客户端控制的管道遍历。

懒惰地处理流可以提高效率；在诸如上面的 filter-map-sum 示例的管道中，过滤、映射和求和可以融合到数据的单次传递中，具有最小的中间状态。懒惰还允许在没有必要时避免检查所有数据。对于诸如“查找第一个超过 1000 个字符的字符串”之类的操作，只需检查足够多的字符串即可找到具有所需特征的字符串，而无需检查源中可用的所有字符串。 （当输入流是无限的而不仅仅是大时，这种行为变得更加重要。）

中间操作又分为无状态操作和有状态操作。无状态操作，例如filter和map ，在处理新元素时不保留先前看到的元素的状态——每个元素都可以独立于其他元素的操作进行处理。有状态的操作，例如 distinct和sorted ，在处理新元素时可能会合并来自先前看到的元素的状态。
有状态的操作可能需要在产生结果之前处理整个输入。例如，在查看流的所有元素之前，无法通过对流进行排序产生任何结果。因此，在并行计算下，一些包含有状态中间操作的管道可能需要对数据进行多次传递，或者可能需要缓冲重要数据。仅包含无状态中间操作的管道可以单次处理，无论是顺序的还是并行的，数据缓冲最少。
此外，一些操作被认为是短路操作。如果在呈现无限输入时，中间操作可能会产生有限流，则它是短路的。如果一个终端操作在有无限输入时可能会在有限时间内终止，那么它就是短路的。在管道中进行短路操作是无限流处理在有限时间内正常终止的必要条件，但不是充分条件。

并行性

具有显式for-循环的处理元素本质上是串行的。流通过将计算重新定义为聚合操作的管道，而不是作为对每个单独元素的命令式操作来促进并行执行。所有流操作都可以串行或并行执行。 JDK 中的流实现创建串行流，除非明确请求并行性。例如，
Collection有方法java.util.Collection.stream和 java.util.Collection.parallelStream ，它们分别产生顺序流和并行流；其他承载流的方法，例如IntStream.range(int, int)产生顺序流，但这些流可以通过调用它们的BaseStream.parallel()方法有效地并行化。要并行执行先前的“小部件权重总和”查询，我们将执行以下操作：

```
int sumOfWeights = widgets.parallelStream()
            .filter(b -> b.getColor() == RED)
            .mapToInt(b -> b.getWeight())
            .sum();
```

这个例子的串行和并行版本之间的唯一区别是初始流的创建，使用“ parallelStream() ”而不是“ stream()
”。启动终端操作时，流管道会根据调用它的流的方向顺序或并行执行。流是串行执行还是并行执行可以使用isParallel()方法确定，并且可以使用BaseStream.sequential()和 BaseStream.parallel()
操作修改流的方向。启动终端操作时，流管道会根据调用它的流的模式顺序或并行执行。 除了被明确识别为非确定性的操作（例如findAny()外，流是顺序执行还是并行执行不应改变计算结果。 大多数流操作接受描述用户指定行为的参数，这些参数通常是
lambda 表达式。为了保持正确的行为，这些行为参数必须是无干扰的，并且在大多数情况下必须是无状态的。此类参数始终是函数接口的实例，例如java.util.function.Function ，并且通常是 lambda 表达式或方法引用。

不干涉

流使您能够对各种数据源执行可能并行的聚合操作，甚至包括ArrayList等非线程安全的集合。只有在流管道执行期间我们可以防止对数据源的干扰，这才有可能。除了 escape-hatch 操作iterator()和spliterator()
，执行在调用终端操作时开始，并在终端操作完成时结束。对于大多数数据源来说，防止干扰意味着确保数据源在流管道执行过程中完全不被修改。值得注意的例外是其源是并发集合的流，这些集合专门设计用于处理并发修改。并发流源是那些Spliterator报告CONCURRENT特征的源。
因此，源可能不是并发的流管道中的行为参数永远不应修改流的数据源。如果行为参数修改或导致修改流的数据源，则称该行为参数会干扰非并发数据源。不干扰的需求适用于所有管道，而不仅仅是并行管道。除非流源是并发的，否则在流管道执行期间修改流的数据源可能会导致异常、不正确的答案或不一致的行为。对于表现良好的流源，可以在终端操作开始之前修改源，这些修改将反映在覆盖的元素中。例如，考虑以下代码：

```
List<String> l = new ArrayList(Arrays.asList("one", "two"));
Stream<String> sl = l.stream();
l.add("three");
String s = sl.collect(joining(" "));
```

首先创建一个包含两个字符串的列表：“one”；和“二”。然后从该列表创建一个流。接下来通过添加第三个字符串来修改列表：“three”。最后，流的元素被收集并连接在一起。由于列表在终端collect操作开始之前被修改，结果将是一个字符串“一二三”。从
JDK 集合返回的所有流以及大多数其他 JDK 类都以这种方式表现良好；对于其他库生成的流，请参阅低级流构建以了解构建行为良好的流的要求。

无状态行为

如果流操作的行为参数是有状态的，则流管道结果可能是不确定的或不正确的。有状态的 lambda（或实现适当功能接口的其他对象）的结果取决于在流管道执行期间可能更改的任何状态。有状态 lambda 的一个示例是map()的参数：

```
Set<Integer> seen=Collections.synchronizedSet(new HashSet<>());
        stream.parallel().map(e->{
        if(seen.add(e))return 0;
        else return e;
        })...
```

在这里，如果映射操作是并行执行的，由于线程调度的差异，相同输入的结果可能因运行而异，而对于无状态 lambda 表达式，结果将始终相同。
另请注意，尝试从行为参数访问可变状态会给您带来安全和性能方面的错误选择；如果您不同步对该状态的访问，您就会遇到数据竞争，因此您的代码会被破坏，但如果您确实同步对该状态的访问，您可能会面临竞争破坏您正在寻求从中受益的并行性的风险。最好的方法是避免有状态的行为参数完全流式操作；通常有一种方法可以重组流管道以避免有状态。

副作用

通常不鼓励对流操作的行为参数产生副作用，因为它们通常会导致无意中违反无状态要求以及其他线程安全隐患。
如果行为参数确实有副作用，除非明确说明，否则不能保证这些副作用对其他线程的可见性，也不能保证对同一流管道内“相同”元素的不同操作在同一个线程中执行。此外，这些影响的排序可能令人惊讶。即使管道被约束以产生与流源的遇到顺序一致的结果

（例如， IntStream.range(0,5).parallel().map(x -> x*2).toArray()必须产生[0, 2, 4, 6, 8] ）
，不保证映射器函数应用于单个元素的顺序，或在哪个线程中为给定元素执行任何行为参数。

许多可能倾向于使用副作用的计算可以在没有副作用的情况下更安全有效地表达，例如使用归约而不是可变累加器。但是，使用println()进行调试等副作用通常是无害的。少数流操作，例如forEach()和peek()
，只能通过副作用进行操作；这些应小心使用。

作为如何将不恰当地使用副作用的流管道转换为不使用副作用的流管道的示例，以下代码在字符串流中搜索匹配给定正则表达式的字符串，并将匹配项放入列表中。

```
ArrayList<String> results=new ArrayList<>();
        stream.filter(s->pattern.matcher(s).matches())
        .forEach(s->results.add(s));
```

此代码不必要地使用了副作用。如果并行执行， ArrayList的非线程安全性会导致不正确的结果，并且添加所需的同步会导致争用，从而破坏并行性的好处。此外，在这里使用副作用是完全没有必要的；forEach()
可以简单地替换为更安全、更高效且更易于并行化的归约操作：

```
 List<String>results=stream.filter(s->pattern.matcher(s).matches())
        .collect(Collectors.toList());
```

订购

流可能有也可能没有定义的遭遇顺序。流是否有遇到顺序取决于源和中间操作。某些流源（例如List或数组）本质上是有序的，而其他流源（例如HashSet ）则不是。一些中间操作，例如sorted()
，可能会在其他无序的流上施加遇到顺序，而其他操作可能会呈现无序的有序流，例如BaseStream.unordered() 。此外，一些终端操作可能会忽略遇到顺序，例如forEach() 。

如果流是有序的，则大多数操作都被限制为按照遇到的顺序对元素进行操作；如果流的源是包含[1, 2, 3]的List ，则执行map(x -> x*2)的结果必须是[2, 4, 6] 。但是，如果源没有定义的遇到顺序，那么值[2, 4, 6]
的任何排列都是有效的结果。

对于顺序流，遇到顺序的存在与否不会影响性能，只会影响确定性。如果流是有序的，在相同的源上重复执行相同的流管道将产生相同的结果；如果没有排序，重复执行可能会产生不同的结果。

对于并行流，放宽排序约束有时可以提高执行效率。如果元素的排序不相关，则可以更有效地实现某些聚合操作，例如过滤重复项 （ distinct() ） 或分组缩减 （ Collectors.groupingBy() ）。类似地，本质上与遇到顺序相关的操作，例如limit() ，可能需要缓冲以确保正确排序，从而破坏并行性的好处。在流有遇到顺序的情况下，但用户并不特别关心该遇到顺序，使用unordered()显式地对流进行降序可能会提高某些有状态或终端操作的并行性能。然而，大多数流管道，例如上面的“块的权重之和”示例，即使在排序约束下仍然有效地并行化。

减少操作

归约操作（也称为折叠）采用一系列输入元素并通过重复应用组合操作将它们组合成单个汇总结果，例如找到一组数字的总和或最大值，或将元素累加到列表中.流类具有多种形式的通用归约操作，称为reduce()和collect()，以及多种专门的归约形式，例如sum() 、 max()或count() 。 当然，这样的操作可以很容易地实现为简单的顺序循环，如下所示：

```
int sum=0;
        for(int x:numbers){
        sum+=x;
        }
```

但是，有充分的理由更喜欢减少操作而不是上述的变异累积。归约不仅“更抽象”—它作为一个整体而不是单个元素对流进行操作—而且正确构造的归约操作本质上是可并行的，只要用于处理元素的函数是关联的和无国籍的。例如，给定一个我们想要求和的数字流，我们可以这样写：

```
int sum=numbers.stream().reduce(0,(x,y)->x+y);
```

或者：

```
int sum=numbers.stream().reduce(0,Integer::sum);
```

这些归约操作可以安全地并行运行，几乎不需要修改：

```
int sum=numbers.parallelStream().reduce(0,Integer::sum);
```

归约可以很好地并行化，因为实现可以并行地对数据的子集进行操作，然后结合中间结果得到最终的正确答案。

（即使该语言具有“并行for-each”结构，可变累加方法仍需要开发人员为共享累加变量sum提供线程安全更新，并且所需的同步可能会消除并行性带来的任何性能增益.） 使用reduce()代替了并行化归约操作的所有负担，并且库可以提供有效的并行实现，而无需额外的同步。

前面显示的“小部件”示例显示了缩减如何与其他操作结合使用批量操作替换 for 循环。如果widgets是Widget对象的集合，它们有一个getWeight方法，我们可以通过以下方式找到最重的小部件

```
OptionalInt heaviest=widgets.parallelStream()
        .mapToInt(Widget::getWeight)
        .max();
```

在其更一般的形式中，对类型元素的reduce操作 产生类型的结果需要三个参数：

```
<U> U reduce(U identity,
      BiFunction<U, ? super T,U>accumulator,
      BinaryOperator<U> combiner
      );
```

这里，恒等元素既是归约的初始种子值，也是没有输入元素时的默认结果。累加器函数获取部分结果和下一个元素，并产生一个新的部分结果。 combiner函数组合两个部分结果以产生新的部分结果。（合并器在并行缩减中是必需的，其中输入被分区，为每个分区计算部分累积，然后将部分结果组合以产生最终结果。）

更正式地说， identity值必须是组合器函数的标识。这意味着对于所有u ，combiner.apply(identity, u)等于u 。此外，combiner函数必须是关联的，并且必须与accumulator函数兼容：对于所有u和t
， combiner.apply(u, accumulator.apply(identity, t))必须equals()到accumulator.apply(u, t) 。三参数形式是二参数形式的推广，将映射步骤合并到累加步骤中。我们可以使用更一般的形式重新转换简单的权重总和示例，如下所示：

```
int sumOfWeights = widgets.stream()
.reduce(0,(sum, b) -> sum + b.getWeight()) Integer::sum);
```

尽管显式 map-reduce 形式更具可读性，因此通常应该是首选。广义形式适用于可以通过将映射和归约组合到单个函数来优化重要工作的情况。

可变归约

可变归约操作在处理流中的元素时将输入元素累积到可变结果容器中，例如Collection或StringBuilder 。 如果我们想获取一个字符串流并将它们连接成一个长字符串，我们可以通过普通的归约来实现：

```
String concatenated = strings.reduce("", String::concat)
```

我们会得到想要的结果，它甚至可以并行工作。但是，我们可能对性能不满意！这样的实现会进行大量的字符串复制，并且运行时间将是O(n^2)
的字符数。一种更高效的方法是将结果累积到StringBuilder中，这是一个用于累积字符串的可变容器。我们可以使用与普通归约相同的技术来并行化可变归约。

可变归约操作称为collect() ，因为它将所需的结果收集到一个结果容器中，例如Collection 。一个collect操作需要三个函数：一个用于构造结果容器的新实例的供应商函数，一个将输入元素合并到结果容器中的累加器函数，以及将一个结果容器的内容合并到另一个容器中的组合函数。 this 的形式与普通归约的一般形式非常相似：

```
<R> R collect(Supplier<R> supplier,BiConsumer<R, ? super T>
accumulator,BiConsumer<R, R> combiner);
```

与reduce()一样，以这种抽象方式表达collect的一个好处是它可以直接进行并行化：我们可以并行累加部分结果，然后将它们组合起来，只要累加和组合函数满足适当的要求。例如，要将流中元素的String
表示形式收集到ArrayList中，我们可以编写明显的顺序 for-each 形式：

```
ArrayList<String> strings = new ArrayList<>();
for (T element : stream) {
    strings.add(element.toString());
}
```

或者我们可以使用一个可并行化的收集表单：

```
 ArrayList<String> strings=stream.collect(()->new ArrayList<>(),(c,e)-
>c.add(e.toString()),(c1,c2)->c1.addAll(c2));
```

或者，将映射操作从累加器函数中提取出来，我们可以更简洁地表示为：

```
List<String> strings = stream.map(Object::toString).collect(ArrayList::new,
ArrayList::add, ArrayList::addAll);
```

在这里，我们的供应商只是ArrayList constructor ，累加器将字符串化元素添加到ArrayList中，组合器只需使用addAll将字符串从一个容器复制到另一个容器。

collect的三个方面—供应商、累加器和组合器—是紧密耦合的。我们可以使用Collector的抽象来捕获所有三个方面。上面用于将字符串收集到List中的示例可以使用标准Collector重写为：

```
List<String> strings = stream.map(Object::toString).collect(Collectors.toList());
```

将可变归约打包到 Collector 中还有另一个优点：可组合性。 Collectors类包含许多预定义的收集器工厂，包括将一个收集器转换为另一个收集器的组合器。例如，假设我们有一个收集器来计算员工流的工资总和，如下所示：

```
Collector<Employee, ?, Integer> summingSalaries =
Collectors.summingInt(Employee::getSalary);
```

（第二个类型参数的?仅仅表明我们不关心这个收集器使用的中间表示。）如果我们想创建一个收集器来按部门列出工资总和，我们可以使用groupingBy重用summingSalaries ：

```
Map<Department, Integer> salariesByDept  =
employees.stream().collect(Collectors.groupingBy(Employee::getDepartment,
                                                 summingSalaries));
```

与常规归约操作一样， collect()操作只有在满足适当条件时才能并行化。对于任何部分累积的结果，将其与空结果容器组合必须产生等效结果。也就是说，对于作为任何一系列累加器和组合器调用的结果的部分累加结果p ，
p必须等价于combiner.apply(p, supplier.get()) 。

此外，无论计算如何拆分，它都必须产生等效的结果。对于任何输入元素t1和t2 ，以下计算中的结果r1和r2必须相等：

```
A a1 = supplier.get();
accumulator.accept(a1, t1);
accumulator.accept(a1, t2);
R r1 = finisher.apply(a1);  // result without splitting
A a2 = supplier.get();
accumulator.accept(a2, t1);
A a3 = supplier.get();
accumulator.accept(a3, t2);
R r2 = finisher.apply(combiner.apply(a2, a3));  // result with splitting
```

这里，等价通常意味着根据Object.equals(Object) 。但在某些情况下，等价可能会放宽以解释顺序上的差异。

约简、并发和排序

使用一些复杂的归约操作，例如产生Map的collect() ，例如：

```
Map<Buyer, List<Transaction>> salesByBuyer  =
txns.parallelStream().collect(Collectors.groupingBy(Transaction::getBuyer));
```

并行执行操作实际上可能适得其反。这是因为组合步骤（通过键将一个Map合并到另一个Map）对于某些Map实现来说可能是昂贵的。然而，假设在这个归约中使用的结果容器是一个可并发修改的集合—例如java.util.concurrent.ConcurrentHashMap。

在这种情况下，累加器的并行调用实际上可以将它们的结果同时存储到同一个共享结果容器中，从而消除了组合器合并不同结果容器的需要。这可能会提高并行执行性能。我们称之为并发减少。支持并发归约的Collector.Characteristics.CONCURRENTCollector标记。但是，并发收集也有一个缺点。如果多个线程同时将结果存放到共享容器中，则存放结果的顺序是不确定的。因此，仅当排序对于正在处理的流不重要时，才可能进行并发减少。
Stream.collect(Collector)

实现只会执行并发减少，如果流是并行的；

收集器具有Collector.Characteristics.CONCURRENT特征，并且；流是无序的，或者收集器具有Collector.Characteristics.UNORDERED特性。

您可以使用BaseStream.unordered()方法确保流是无序的。例如：

```
Map<Buyer, List<Transaction>> salesByBuyer =
txns.parallelStream().unordered().collect(groupingByConcurrent(Transaction::getBuyer));
```

（其中Collectors.groupingByConcurrent是groupingBy的并发等价物）。

请注意，如果给定键的元素按照它们在源中出现的顺序出现很重要，那么我们不能使用并发归约，因为排序是并发插入的牺牲品之一。然后，我们将受限于实现顺序归约或基于合并的并行归约。

关联性

如果满足以下条件，则运算符或函数op是关联的：

```
(a op b) op c == a op (b op c)
```

如果我们将其扩展到四个术语，则可以看出这对并行评估的重要性：

```
a op b op c op d == (a op b) op (c op d)
```

所以我们可以并行评估(a op b)和(c op d) ，然后在结果上调用op 。

关联运算的示例包括数字加法、最小值和最大值以及字符串连接。

低级流构建

到目前为止，所有流示例都使用java.util.Collection.stream()或java.util.Arrays.stream(Object[])等方法来获取流。这些流承载方法是如何实现的?

StreamSupport类有许多用于创建流的低级方法，它们都使用某种形式的java.util.Spliterator 。 spliterator 是java.util.Iterator的并行类似物；它描述了一个（可能是无限的）元素集合，支持顺序推进、批量遍历，以及将输入的某些部分拆分到另一个可以并行处理的拆分器中。在最低级别，所有流都由拆分器驱动。

在实现拆分器时有许多实现选择，几乎所有这些选择都是在实现的简单性和使用该拆分器的流的运行时性能之间进行权衡。创建拆分器的最简单但性能最低的方法是使用java.util.Spliterators.spliteratorUnknownSize(java.util.Iterator, int)从迭代器创建一个。虽然这样的拆分器会起作用，但它可能会提供较差的并行性能，因为我们丢失了大小信息（底层数据集有多大），并且受限于简单的拆分算法。

更高质量的拆分器将提供平衡且已知大小的拆分、准确的大小信息以及拆分器或数据的许多其他characteristics ，实现可用于优化执行。

可变数据源的拆分器有一个额外的挑战；绑定到数据的时间，因为数据可能会在创建拆分器的时间和执行流管道的时间之间发生变化。理想情况下，流的拆分器将报告IMMUTABLE或CONCURRENT的特征；如果不是，它应该是后期绑定。如果源不能直接提供推荐的拆分器，它可以使用Supplier间接提供拆分器，并通过Supplier接受版本的stream()构造流。只有在流管道的终端操作开始后，才能从供应商处获得拆分器。

这些要求显着减少了流源突变和流管道执行之间的潜在干扰范围。基于具有所需特征的拆分器的流，或使用基于供应商的工厂表单的流，在终端操作开始之前不受数据源修改的影响（前提是流操作的行为参数满足非干涉和无国籍状态）。有关详细信息，请参阅不干扰